Reverse Engineering Malware with Automated Scripts

What we found in malware analysis so far

- From surface analysis:
    - Its file type is a dynamic link library (DLL).
    - It has many export functions such as "GnrkQr". We also found that the malware was set to launch with the string "GnrkQr" by examining the output of Autoruns.
    - It imports only 4 modules: KERNEL32.dll, USER32.dll, GDI32.dll, and ADVAPI32.dll.
- From dynamic analysis:
    - It tries to connect to "tyo1964.com:443" automatically. The server seems to be a command and control server.
    - The traffic to "tyo1964.com:443" contains characteristic strings. For example, its http header contains "X-Session", "X-Status", "X-Size" and "X-Sn".
    - It uses the victim organization's private proxy server.

What we have not found yet

- We have not revealed the malware's capabilities that the attacker could do in the incident.
- We have not known where the malware leaves evidence for execution of those capabilities on.

- If we find one of those at least, we can enhance the analysis result.
  - For example, a certain RAT has a file uploading function to an infected host. When a file is uploaded, the malware creates a file that the name ends with ".tmp" temporary with the original file. If you know this by reverse engineering, you can discover the file that the attacker has sent by analyzing the NTFS journal file with forensic analysis.
- To achieve such purposes, we should perform reverse engineering.

What should we do next?

- We will examine APIs that the malware uses, and investigate how the malware uses them to reveal its capabilities.
- We will find characteristic strings by examining strings contained in the malware. Investigating those strings helps us to get such as mutex, paths of related files, and protocol headers that might be effective evidence. And we often identify the name of the malware from those evidence.

Reverse engineering tools

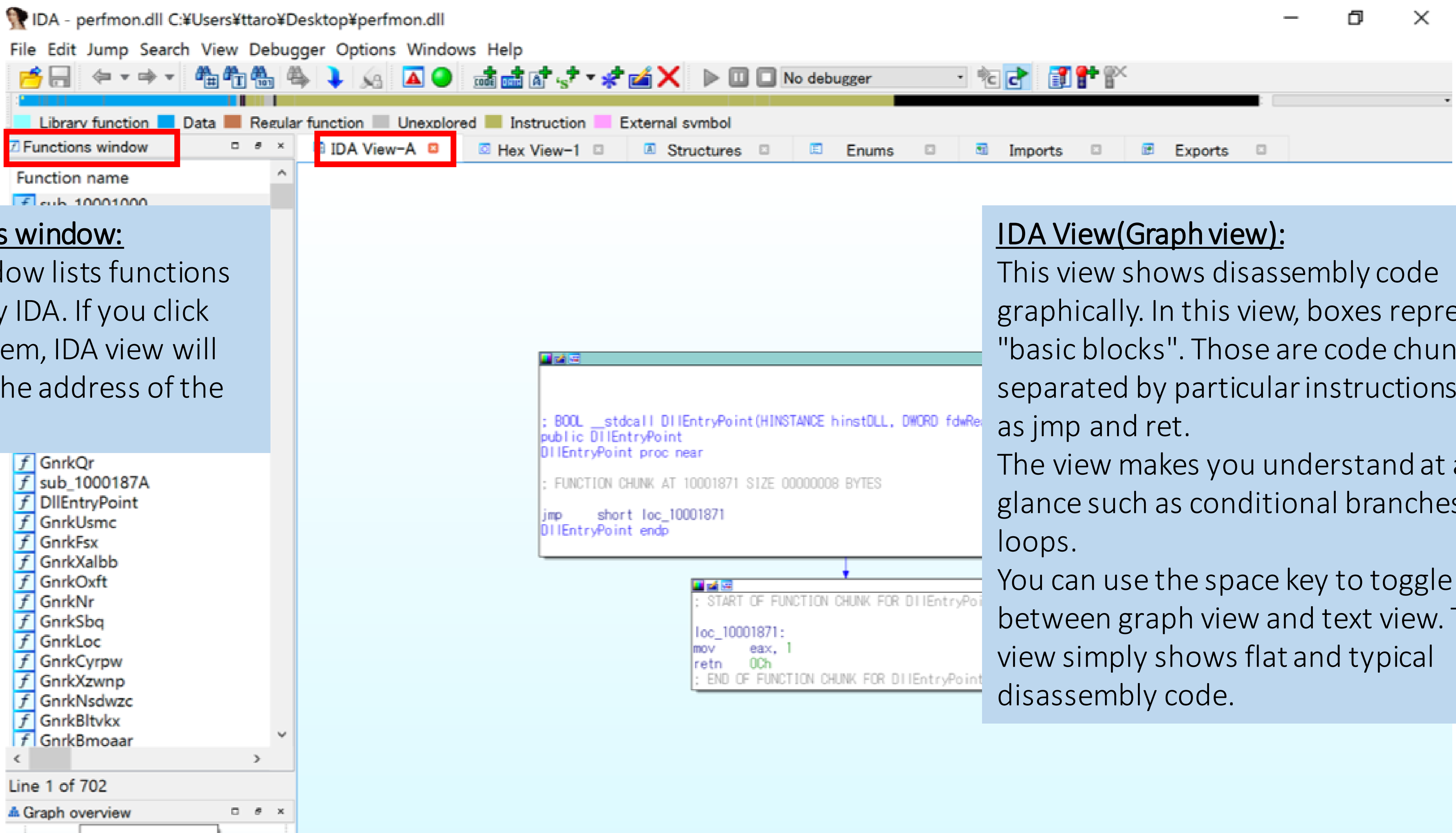
- IDA is the de facto standard tool for reverse engineering.
- We will investigate the malware using IDA's scripting capabilities.
- If you do not have a valid copy of IDA Pro, you can use IDA Evaluation Version for exercises.

https://www.hex-rays.com/products.shtml

Opening the malware with IDA (1)

- Drag the malware and drop to the IDA color icon idaq.exe on your desktop of the analysis VM for opening.

Opening the malware with IDA (2)



sub_10001000
GnrkQr
sub_1000187A
DllEntryPoint
GnrkUsmc
GnrkFsx
GnrkXalbb
GnrkOxft
GnrkNr
GnrkSbq
GnrkLoc
GnrkCyrpw
GnrkXzwnp
GnrkNsdwzc
GnrkBltvkx
GnrkBmoaar

```
; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwRe
public DllEntryPoint
DllEntryPoint proc near

; FUNCTION CHUNK AT 10001871 SIZE 00000008 BYTES

jmp     short loc_10001871
DllEntryPoint endp
```

```
; START OF FUNCTION CHUNK FOR DllEntryPoi

loc_10001871:
mov     eax, 1
retn    0Ch
; END OF FUNCTION CHUNK FOR DllEntryPoint
```

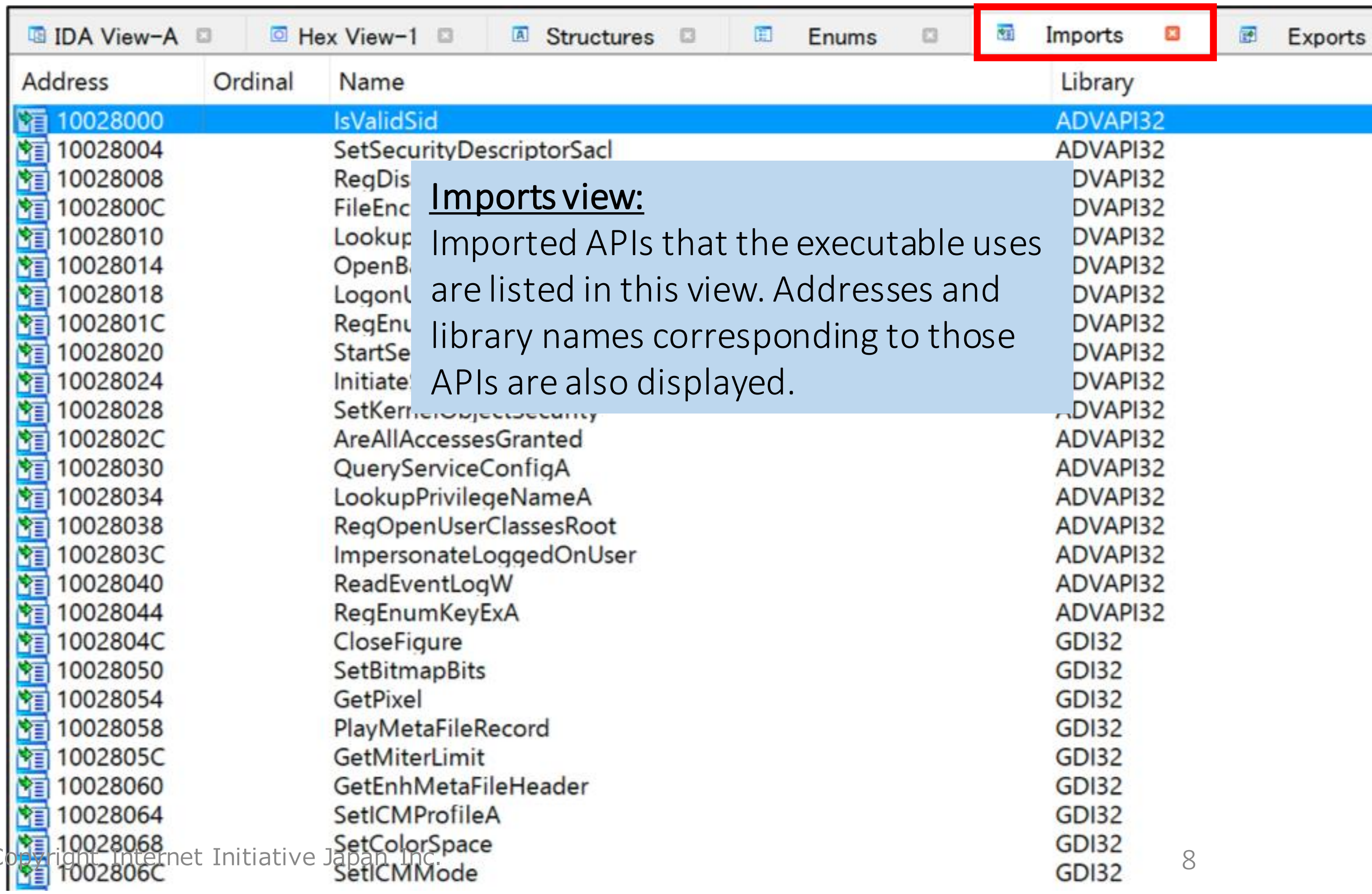
Functions window:
This window lists functions parsed by IDA. If you click one of them, IDA view will jump to the address of the function.

IDA View(Graph view):
This view shows disassembly code graphically. In this view, boxes represent "basic blocks". Those are code chunks separated by particular instructions such as jmp and ret.
The view makes you understand at a glance such as conditional branches and loops.
You can use the space key to toggle between graph view and text view. Text view simply shows flat and typical disassembly code.

Examining imported APIs

- Click the "Imports view" tab.

- We have already known APIs included in the malware by performing surface analysis though, we examine this for the purpose of getting used to general IDA operations.



Address	Ordinal	Name	Library
10028000		IsValidSid	ADVAPI32
10028004		SetSecurityDescriptorSacl	ADVAPI32
10028008		RegDis	DVAPI32
1002800C		FileEnc	DVAPI32
10028010		Lookup	DVAPI32
10028014		OpenB	DVAPI32
10028018		Logon	DVAPI32
1002801C		RegEnu	DVAPI32
10028020		StartSe	DVAPI32
10028024		Initiate	DVAPI32
10028028		SetKern	ADVAPI32
1002802C		AreAllAccessesGranted	ADVAPI32
10028030		QueryServiceConfigA	ADVAPI32
10028034		LookupPrivilegeNameA	ADVAPI32
10028038		RegOpenUserClassesRoot	ADVAPI32
1002803C		ImpersonateLoggedOnUser	ADVAPI32
10028040		ReadEventLogW	ADVAPI32
10028044		RegEnumKeyExA	ADVAPI32
1002804C		CloseFigure	GDI32
10028050		SetBitmapBits	GDI32
10028054		GetPixel	GDI32
10028058		PlayMetaFileRecord	GDI32
1002805C		GetMiterLimit	GDI32
10028060		GetEnhMetaFileHeader	GDI32
10028064		SetICMProfileA	GDI32
10028068		SetColorSpace	GDI32
1002806C		SetICMMode	GDI32

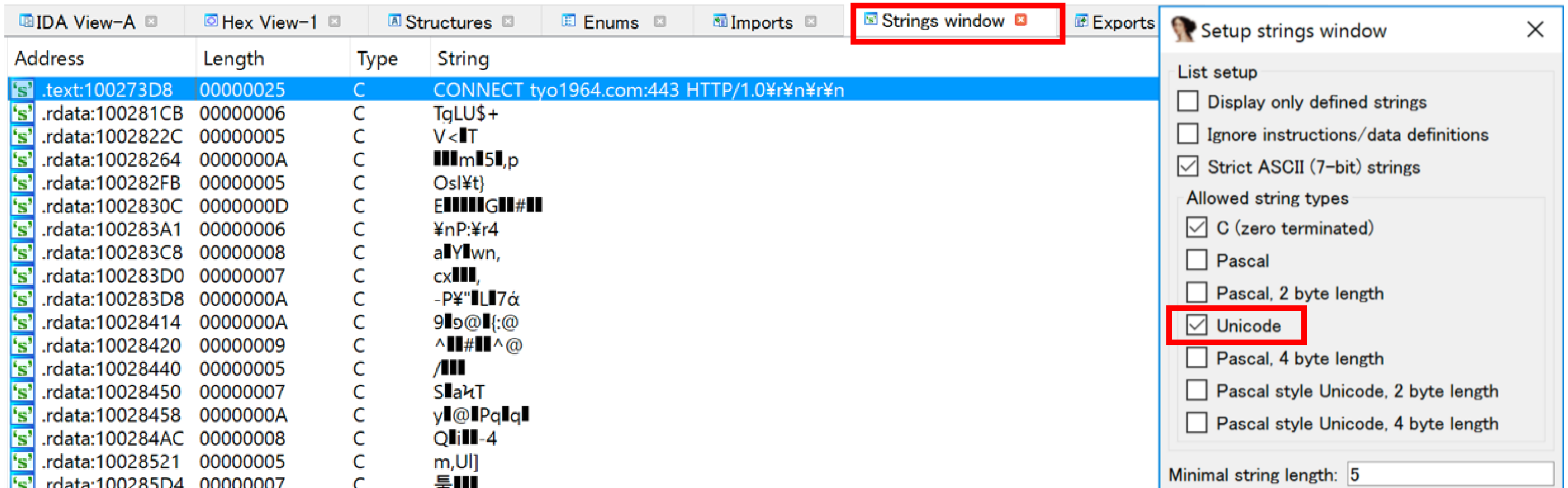
Imports view:
Imported APIs that the executable uses are listed in this view. Addresses and library names corresponding to those APIs are also displayed.

The results of the examining imported functions

- The "Imports view" tab shows:
  - Only four libraries are imported: KERNEL32, USER32, GDI32 and ADVAPI32.
  - This looks strange. Typically, Windows applications that can communicate with TCP/IP require one or more libraries loaded that are WS2_32, WinHttp, and WinINet.
  - Therefore, the malware author might use some tricks to hide libraries.

Viewing raw strings

- Press the Shift+F12 keys to open the strings window.
- By default, it shows only C-style strings. To view unicode strings, right-click and choose "Setup...", then check "Unicode".
- A C-style string is an array of characters that uses null-terminator (\0).



Address	Length	Type	String
.text:100273D8	00000025	C	CONNECT tyo1964.com:443 HTTP/1.0¥r¥n¥r¥n
.rdata:100281CB	00000006	C	TqLU$+
.rdata:1002822C	00000005	C	V<█T
.rdata:10028264	0000000A	C	███m█5█,p
.rdata:100282FB	00000005	C	Osl¥t}
.rdata:1002830C	0000000D	C	E█████G██#██
.rdata:100283A1	00000006	C	¥nP:¥r4
.rdata:100283C8	00000008	C	a█Y█wn,
.rdata:100283D0	00000007	C	cx███,
.rdata:100283D8	0000000A	C	-P¥"█L█7ά
.rdata:10028414	0000000A	C	9█ɒ@█{:@
.rdata:10028420	00000009	C	^██#██^@
.rdata:10028440	00000005	C	/███
.rdata:10028450	00000007	C	S█aꝁT
.rdata:10028458	0000000A	C	y█@█Pq█q█
.rdata:100284AC	00000008	C	Q█i██-4
.rdata:10028521	00000005	C	m,Ul]
.rdata:100285D4	00000007	C	둨███

Setup strings window

List setup
- [ ] Display only defined strings
- [ ] Ignore instructions/data definitions
- [x] Strict ASCII (7-bit) strings

Allowed string types
- [x] C (zero terminated)
- [ ] Pascal
- [ ] Pascal, 2 byte length
- [x] Unicode
- [ ] Pascal, 4 byte length
- [ ] Pascal style Unicode, 2 byte length
- [ ] Pascal style Unicode, 4 byte length

Minimal string length: 5

The results of the viewing raw strings

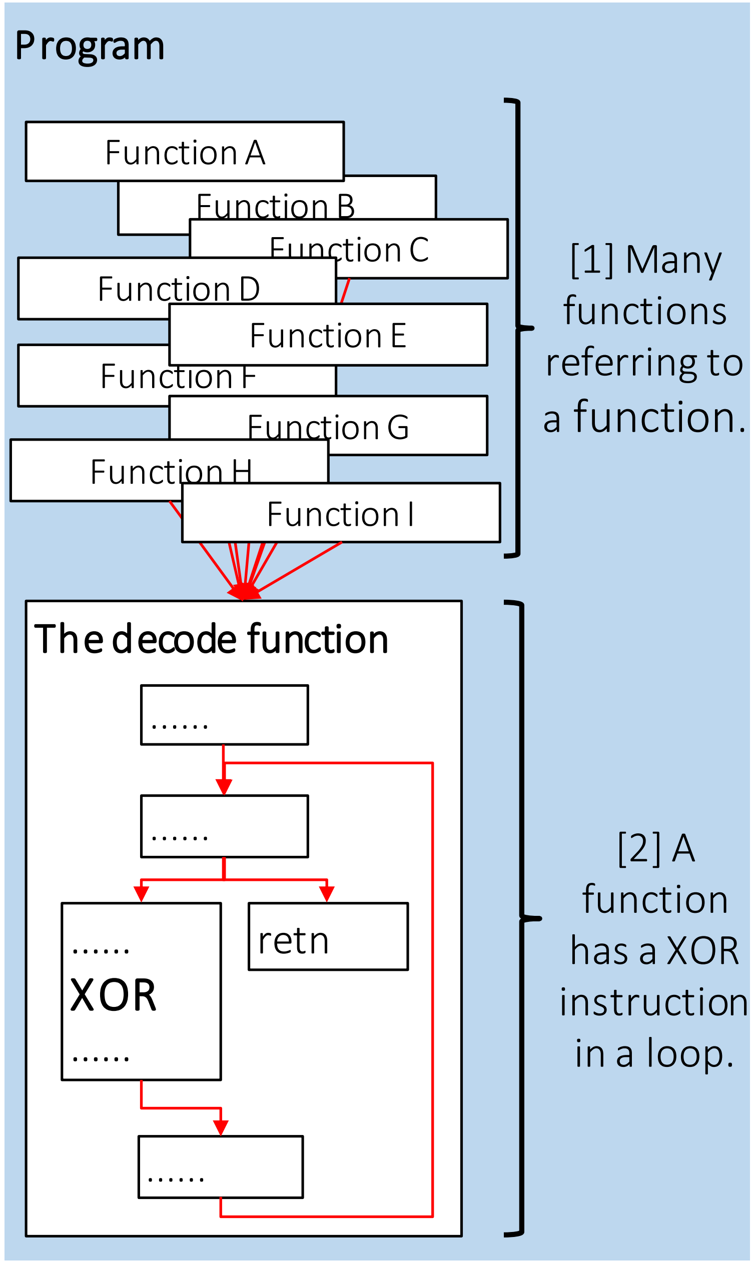
- The "Strings window" shows:
  - We were able to find only a few strings related to HTTP headers, methods, and query parameters.
  - Because of that, the malware author might use some tricks to hide strings.

We have found a new issue

- By examining imported functions and raw strings:
  - API names and HTTP related strings that are actually used in the malware did not appear.
- Why？
  - Are important characters obfuscated?
  - Is the malware packed? – Packing means compression and/or encryption of executable code.
- Let's find the decode function on the assumption that important strings are obfuscated.

The strategy to expose tricks

- The typical decode function have characteristics of:
  1. Being referenced from a lot of functions
     - Whenever the program uses obfuscated strings, the decode function is called.
  2. A XOR loop in the function
     - Using XOR is a established way of implementing encryption and obfuscation including major encryption algorithms.
- Let's find the function that has both of above.
- We try this way manually to understand the logic here. Later, we will introduce several tools such as FLOSS and IDA Scope for automating this method.



Finding the decode function with IDC

- IDC: A C-like scripting language for automating IDA.
- IDAPython: Python bindings for IDA. This is not available for evaluation version of IDA.
- Reference and tutorials:
  - https://www.hex-rays.com/products/ida/support/idadoc/162.shtml
  - https://hex-rays.com/products/ida/support/tutorials/idc/index.shtml
- How to execute:
  - There are two ways to execute IDC scripts. At this time, we use former one.
  - Save your script as text file, then press the Alt+F7 keys on IDA to load and execute the script file.
  - Press the Shift+F2 keys on IDA to open the "Script Execute" window, then write and execute your script on the window.
- We prepared sample scripts. Let's unzip "idc.zip" on your desktop.
- We also prepared scripts written in IDAPython as "idapython.zip". If you have a valid copy of IDA Pro and are familiar with writing Python, you can use this instead of IDC scripts.

IDC Tutorial [1]: Enumerating functions

- Let's start exercises for using IDC scripts with a easy sample.
- This script enumerates functions with the start and the end of addresses.

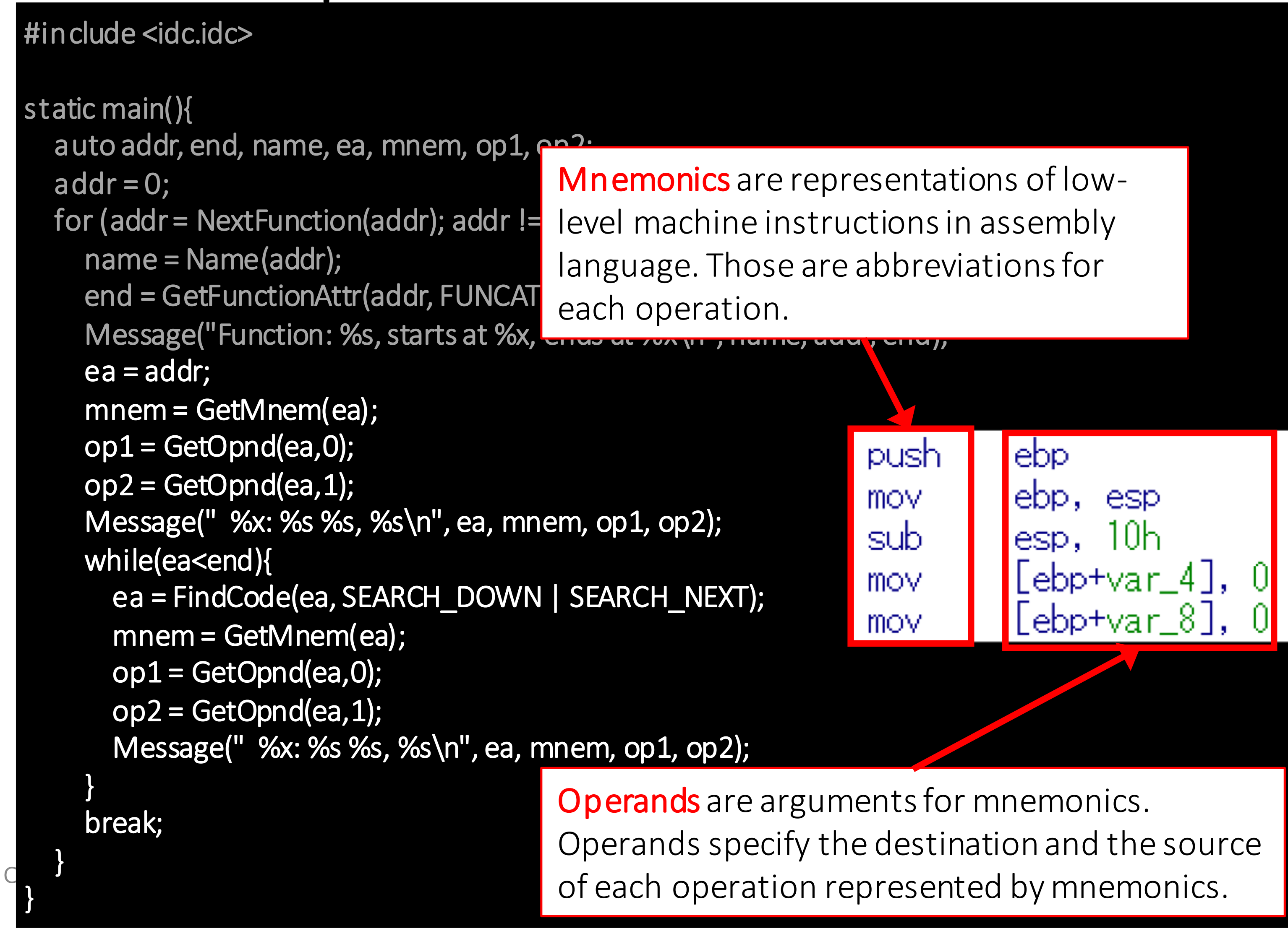
```
#include <idc.idc>

static main(){
    auto addr, end, name, ea;
    addr = 0;
    for (addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr)){
        name = Name(addr);
        end = GetFunctionAttr(addr, FUNCATTR_END);
        Message("Function: %s, starts at %x, ends at %x\n", name, addr, end);
    }
}
```

idclab1.idc

IDC Tutorial [2]: Parsing mnemonics and operands

- This script parses the first function line by line and displays each mnemonic and operand.

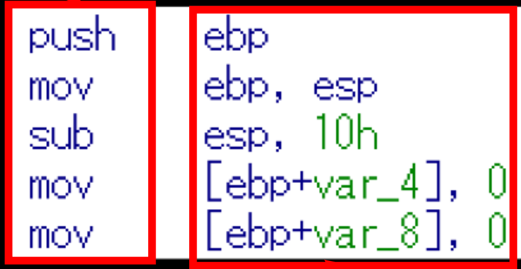


```
#include <idc.idc>

static main(){
   auto addr, end, name, ea, mnem, op1, op2;
   addr = 0;
   for (addr = NextFunction(addr); addr !=
      name = Name(addr);
      end = GetFunctionAttr(addr, FUNCAT
      Message("Function: %s, starts at %x, ends at %x\n", name, addr, end);
      ea = addr;
      mnem = GetMnem(ea);
      op1 = GetOpnd(ea,0);
      op2 = GetOpnd(ea,1);
      Message("  %x: %s %s, %s\n", ea, mnem, op1, op2);
      while(ea<end){
         ea = FindCode(ea, SEARCH_DOWN | SEARCH_NEXT);
         mnem = GetMnem(ea);
         op1 = GetOpnd(ea,0);
         op2 = GetOpnd(ea,1);
         Message("  %x: %s %s, %s\n", ea, mnem, op1, op2);
      }
      break;
   }
}
```

idclab2.idc

Mnemonics are representations of low-level machine instructions in assembly language. Those are abbreviations for each operation.



```
push    ebp
mov     ebp, esp
sub     esp, 10h
mov     [ebp+var_4], 0
mov     [ebp+var_8], 0
```

Operands are arguments for mnemonics. Operands specify the destination and the source of each operation represented by mnemonics.

IDC Tutorial [3]:
Adding colors and changing names

- This script parses the first function line by line and changes colors (#66ccff/light blue) for lines that contain a call instruction. It also changes the name of the function.

idclab3.idc

```
#include <idc.idc>

static main(){
    auto addr, end, name, ea, mnem;
    addr = 0;
    for (addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr)){
        name = Name(addr);
        end = GetFunctionAttr(addr, FUNCATTR_END);
        Message("Function: %s, starts at %x, ends at %x\n", name, addr, end);
        while(ea<end){
            ea = FindCode(ea, SEARCH_DOWN | SEARCH_NEXT);
            mnem = GetMnem(ea);
            if(strstr("call", mnem) == 0) SetColor(ea, CIC_ITEM, 0xffcc66);
        }
        name = sprintf("checked_sub_%x", addr);
        MakeNameEx(addr, name, SN_NOCHECK);
        break;
    }
}
```

Finding XOR instructions [1]

- This script changes colors for lines that contain a XOR instruction and changes names of functions that contain at least one XOR instruction.

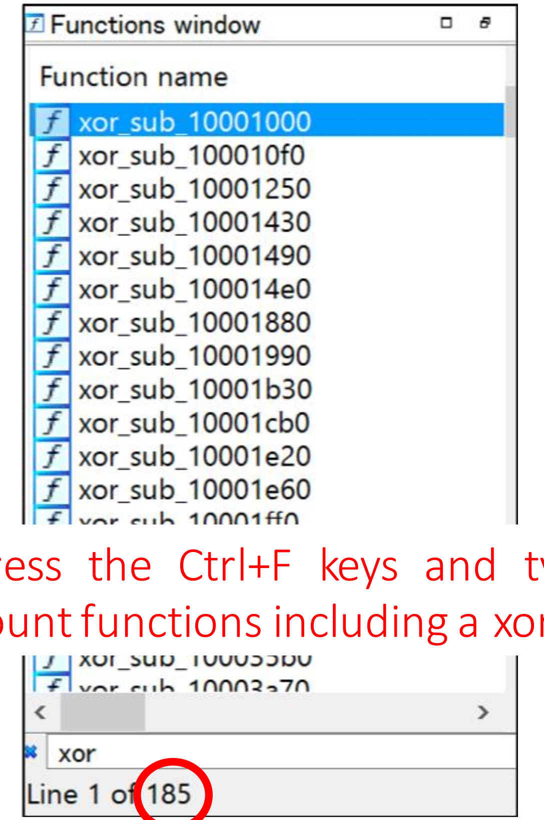
```
#include <idc.idc>

static main(){
    auto addr, end, name, ea, mnem;
    addr = 0;
    for (addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr)){
        name = Name(addr);
        end = GetFunctionAttr(addr, FUNCATTR_END);
        ea = addr;
        while(ea<end){
            ea = FindCode(ea, SEARCH_DOWN | SEARCH_NEXT);
            mnem = GetMnem(ea);
            if(strstr("xor", mnem) == 0){
                SetColor(ea, CIC_ITEM, 0xffcc66);
                name = sprintf("xor_sub_%x", addr);
                MakeNameEx(addr, name, SN_NOCHECK);
                Message("Function: %s has XOR at %x(%s)\n", name, ea);
            }
        }
    }
}
```

idclab4.idc

Finding XOR instructions [2]

- We found 185 functions that contain at least one XOR instruction in the malware.
- But, the purpose of some XOR instructions is to clear a register. Those are not a part of the XOR loop in the decode function. We should filter them out.



Functions window

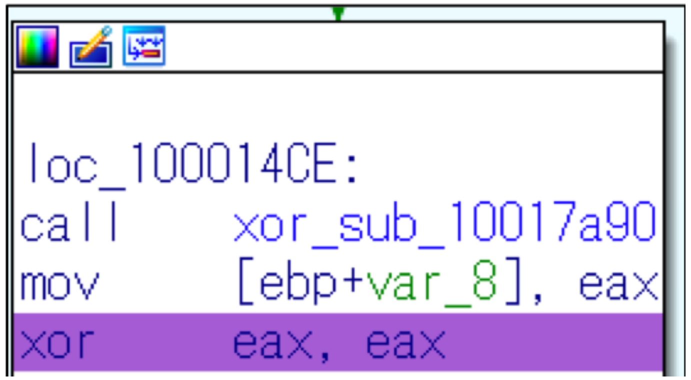
Function name

```
xor_sub_10001000
xor_sub_100010f0
xor_sub_10001250
xor_sub_10001430
xor_sub_10001490
xor_sub_100014e0
xor_sub_10001880
xor_sub_10001990
xor_sub_10001b30
xor_sub_10001cb0
xor_sub_10001e20
xor_sub_10001e60
xor_sub_10001ff0
xor_sub_100033b0
xor_sub_10003a70
```

xor

Line 1 of 185

Press the Ctrl+F keys and type "xor" to count functions including a xor instruction.



```
loc_100014CE:
call    xor_sub_10017a90
mov     [ebp+var_8], eax
xor     eax, eax
```

The XOR instruction with the same operands (e.g. xor eax, eax) means to clear the register to zero.

P	Q	P XOR Q
1	1	0
1	0	1
0	1	1
0	0	0

Finding XOR instructions [3]

- This script changes colors for lines that contain a XOR instruction, changes names of functions that contain at least one XOR instruction, and filters out XOR instructions that clear a register.

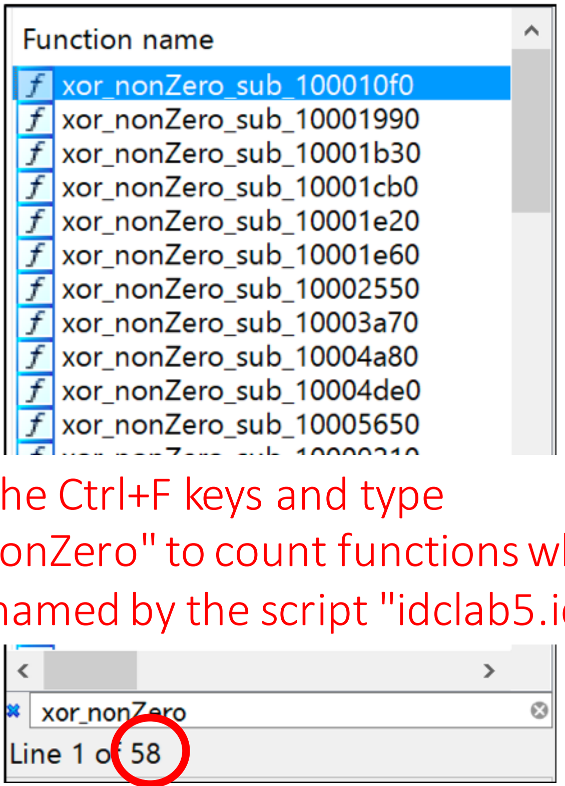
idclab5.idc

```
#include <idc.idc>

static main(){
    auto addr, end, name, ea, mnem, op1, op2;
    addr = 0;
    for (addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr)){
        name = Name(addr);
        end = GetFunctionAttr(addr, FUNCATTR_END);
        ea = addr;
        while(ea<end){
            ea = FindCode(ea, SEARCH_DOWN | SEARCH_NEXT);
            mnem = GetMnem(ea);
            if(strstr("xor", mnem) == 0){
                op1 = GetOpnd(ea,0);
                op2 = GetOpnd(ea,1);
                if(strstr(op1,op2) == 0) continue;
                SetColor(ea, CIC_ITEM, 0xffcc66);
                name = sprintf("xor_nonZero_sub_%x", addr);
                MakeNameEx(addr, name, SN_NOCHECK);
                Message("Function: %s has XOR at %x(%s)\n", name, ea, mnem);
            }
        }
```

Finding XOR instructions [4]

- A number of functions containing a XOR instruction has been 58. But, this is still too many to examine each one.
- We count references for each function called from other functions based on the initial strategy. We can use "cross-references/xrefs" feature of IDA to do that.



Function name

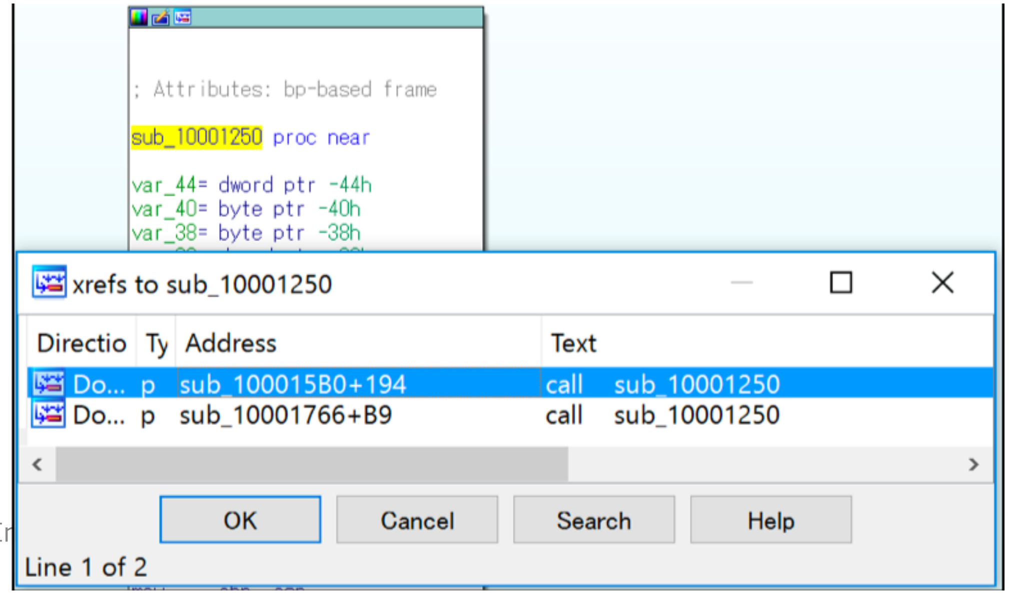
- xor_nonZero_sub_100010f0
- xor_nonZero_sub_10001990
- xor_nonZero_sub_10001b30
- xor_nonZero_sub_10001cb0
- xor_nonZero_sub_10001e20
- xor_nonZero_sub_10001e60
- xor_nonZero_sub_10002550
- xor_nonZero_sub_10003a70
- xor_nonZero_sub_10004a80
- xor_nonZero_sub_10004de0
- xor_nonZero_sub_10005650

xor_nonZero

Line 1 of 58

Press the Ctrl+F keys and type "xor_nonZero" to count functions which are renamed by the script "idclab5.idc".

In the graph view, you can press the X key at the start of a function to display "**cross-references/xrefs**". It shows a list of functions that calls the function you choose.



```
; Attributes: bp-based frame

sub_10001250 proc near

var_44= dword ptr -44h
var_40= byte ptr -40h
var_38= byte ptr -38h
```

xrefs to sub_10001250

Directio	Ty	Address	Text
Do...	p	sub_100015B0+194	call sub_10001250
Do...	p	sub_10001766+B9	call sub_10001250

Line 1 of 2

wizSafe

To be continued…